Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Курсовой проект по курсу

«Операционные системы»

Группа: М8О-209БВ-24

Студент: Андреев М. В.

Преподаватель: Миронов Е.С.

Оценка: _____

Дата: 20.12.25

Москва, 2025

Постановка задачи

Вариант 20.

Исследование 2 аллокаторов памяти: необходимо реализовать два алгоритма аллокации памяти и сравнить их по следующим характеристикам:

- Фактор использования
- Скорость выделения блоков
- Скорость освобождения блоков
- Простота использования аллокатора

Каждый аллокатор памяти должен иметь функции аналогичные стандартным функциям free и malloc.

Перед работой каждый аллокатор инициализируется свободными страницами памяти, выделенными стандартными средствами ядра. Необходимо самостоятельно разработать стратегию тестирования для определения ключевых характеристик аллокаторов памяти. При тестировании нужно свести к минимуму потери точности из-за накладных расходов при измерении ключевых характеристик, описанных выше.

Необходимо сравнить два алгоритма аллокации: списки свободных блоков (наиболее подходящее) и алгоритм Мак-Кьюзи-Кэрелса

Общий метод и алгоритм решения

Использованные системные вызовы:

- **mmap(void addr, size_t length, int prot, int flags, int fd, off_t offset)* – выделяет непрерывный регион виртуальной памяти, возвращает указатель на него.
- **munmap(void addr, size_t length)* – освобождает регион памяти, ранее отображённый через mmap.

Рассмотрим подробнее каждый из алгоритмов, а потом сравним их на различных тестах.

Алгоритм Best Fit (наиболее подходящее)

Алгоритм Best Fit представляет собой классический метод управления динамической памятью, основанный на принципе поиска наиболее подходящего свободного блока для удовлетворения запроса на выделение памяти. Основная цель данного алгоритма — минимизация внешней фрагментации путём выбора блока, размер которого максимально близок к запрашиваемому.

Принцип работы

В основе алгоритма Best Fit лежит структура данных в виде двусвязного списка свободных блоков памяти. Каждый блок содержит заголовок, который хранит метаданные: размер полезной области данных, флаг занятости, а также указатели на предыдущий и следующий свободные блоки.

При инициализации аллокатора с помощью системного вызова mmap выделяется непрерывный регион памяти заданного размера. Весь этот регион

изначально представляется одним большим свободным блоком, который добавляется в список свободных блоков.

Процесс выделения памяти:

1. При получении запроса на выделение памяти размером N, аллокатор выравнивает этот размер до границы машинного слова.
2. К выровненному размеру добавляется размер заголовка блока.
3. Производится линейный поиск по списку свободных блоков для нахождения блока, который удовлетворяет двум условиям:
    - Блок свободен
    - Размер блока больше или равен требуемому
4. Среди всех подходящих блоков выбирается тот, чей размер минимален (наиболее подходящий).
5. Если найденный блок значительно превышает требуемый размер (с учётом минимального размера блока), он разделяется на две части:
    - Первая часть отдаётся пользователю
    - Вторая часть остаётся в списке свободных блоков
6. Выделенный блок помечается как занятый и исключается из списка свободных блоков.

Процесс освобождения памяти:

1. При освобождении блока по указателю, аллокатор вычисляет адрес заголовка блока.
2. Блок помечается как свободный и добавляется в начало списка свободных блоков.
3. Для предотвращения фрагментации выполняется операция слияния (coalescing): если освобождённый блок физически соседствует со свободными блоками в памяти, они объединяются в один большой свободный блок.
4. Объединение происходит рекурсивно до тех пор, пока находятся соседние свободные блоки.

Преимущества и недостатки

Преимущества:

- Минимизация внешней фрагментации за счёт выбора наиболее подходящего блока
- Простота реализации и понятность алгоритма
- Относительно быстрое выделение памяти для средних рабочих нагрузок
- Эффективное использование памяти при правильно настроенном минимальном размере блока

Недостатки:

- Линейная сложность поиска свободного блока (O(n), где n — количество свободных блоков)
- Медленное освобождение памяти из-за необходимости поиска соседних блоков для слияния

- Накопление мелких свободных блоков ("крошек") при интенсивной работе
- Постепенное снижение производительности по мере фрагментации памяти

Особенности реализации

В реализации используется техника быстрого добавления освобождённых блоков в начало списка, что улучшает производительность при повторном выделении недавно освобождённых блоков. Операция слияния соседних блоков выполняется периодически, а не при каждом освобождении, что позволяет сбалансировать производительность и эффективность использования памяти.

Алгоритм Мак-Кьюзи–Кэрелса

Алгоритм Мак-Кьюзи–Кэрелса представляет собой развитие идей аллокаторов, основанных на разбиении памяти по фиксированным классам размеров, и широко применяется в системном программировании, в частности в ядрах семейства BSD. Основная цель данного подхода заключается в уменьшении внутренней фрагментации и накладных расходов при частом выделении и освобождении небольших блоков памяти.

В отличие от аллокатора Best Fit, который использует единый список свободных блоков переменного размера, алгоритм Мак-Кьюзи–Кэрелса организует память как набор страниц фиксированного размера (обычно 4096 байт). Каждая страница может находиться в одном из нескольких состояний: быть полностью свободной, быть разбитой на блоки одинакового размера или являться частью крупного непрерывного блока памяти.

Принцип работы

Аллокатор использует набор предопределённых классов размеров, значения которых обычно являются степенями двойки (например: 16, 32, 64, 128, 256, 512, 1024, 2048 байт). Для каждого класса размеров поддерживается отдельный список страниц, содержащих блоки этого размера.

Процесс выделения памяти:

1. При получении запроса на выделение памяти размером S, аллокатор определяет минимальный класс размеров, который может вместить S.
2. Если размер S меньше максимального класса, производится поиск в списке страниц соответствующего класса:
    - Ищется страница, в которой есть свободные блоки
    - Если такая страница найдена, с помощью битовой карты определяется свободный слот
    - Слот помечается как занятый, и его адрес возвращается пользователю
3. Если все страницы данного класса заняты, выделяется новая свободная страница, разбивается на блоки нужного размера и добавляется в список страниц этого класса.
4. Для больших блоков (превышающих максимальный класс) выделяется несколько последовательных страниц, которые помечаются как единый крупный блок.

Процесс освобождения памяти:

1. По адресу освобождаемого блока определяется страница, которой он принадлежит.
2. Если страница содержит крупный блок, все страницы этого блока возвращаются в пул свободных страниц.
3. Если страница содержит мелкие блоки:
    - В битовой карте страницы соответствующий бит сбрасывается
    - Увеличивается счётчик свободных блоков в странице
    - Если после освобождения страница становится полностью свободной, она возвращается в общий пул свободных страниц

Преимущества и недостатки

Преимущества:

- Высокий фактор использования памяти (до 72% в тестах)
- Минимальная внутренняя фрагментация благодаря классам размеров
- Константное время освобождения памяти
- Эффективная работа с малыми объектами
- Отсутствие заголовков для каждого блока в пределах страницы

Недостатки:

- Медленное выделение памяти из-за необходимости работы с битовыми картами
- Накладные расходы на обслуживание классов размеров
- Потенциальная внутренняя фрагментация внутри классов
- Ограниченный набор размеров блоков

Особенности реализации

В реализации используется компактная битовая карта размером 256 бит (8 × 32 бита), что позволяет отслеживать до 256 блоков на странице. Страницы организованы в односвязные списки по классам размеров. Для больших блоков применяется специальная пометка, позволяющая быстро определить размер блока при освобождении.

Сравнительный анализ подходов

Оба алгоритма используют системные вызовы mmap и munmap для работы с виртуальной памятью, но применяют принципиально разные стратегии управления блоками:

Best Fit:

- Динамическое разбиение памяти
- Переменные размеры блоков
- Акцент на минимизацию внешней фрагментации
- Линейный поиск при выделении
- Слияние соседних блоков при освобождении

Мак-Кьюзи–Кэрелс:

- Статическое разбиение по классам

- Фиксированные размеры блоков внутри страниц
- Акцент на минимизацию внутренней фрагментации
- Быстрый доступ через классы размеров
- Простое освобождение через битовые карты

В следующих разделах будут представлены результаты экспериментального сравнения этих алгоритмов по ключевым характеристикам: скорости выделения и освобождения блоков, фактору использования памяти и простоте реализации.

Код программы

Best_fit.c

```c
#define _GNU_SOURCE
#include <stddef.h>
#include <stdint.h>
#include <string.h>
#include <sys/mman.h>
#include <stdio.h>

// Минимальный размер блока (16 байт данных + заголовок)
#define MIN_BLOCK_SIZE 16
#define HEADER_SIZE sizeof(BlockHeader)

// Структура заголовка блока памяти
typedef struct BlockHeader {
    size_t size;            // Размер данных в блоке (без учета заголовка)
    int is_free;            // Флаг: 1 - свободен, 0 - занят
    struct BlockHeader* next;  // Указатель на следующий блок в списке
    struct BlockHeader* prev;  // Указатель на предыдущий блок
} BlockHeader;

// Структура аллокатора Best Fit
typedef struct {
    void* base;             // Указатель на начало выделенной памяти
    size_t total_size;      // Общий размер выделенной памяти
    BlockHeader* free_list; // Список свободных блоков
    BlockHeader* used_list; // Список занятых блоков (для быстрого доступа)
} BestFitAllocator;

// Функция выравнивания размера до границы слова
static inline size_t align_size(size_t size) {
    return (size + sizeof(void*) - 1) & ~(sizeof(void*) - 1);
}

// Удаление блока из списка (оптимизированная версия)
static void remove_block(BlockHeader** list, BlockHeader* block) {
    if (block->prev) {
        block->prev->next = block->next;
    } else {
        *list = block->next;
    }
```

```
    if (block->next) {
        block->next->prev = block->prev;
    }
}

// Вставка блока в начало списка
static void insert_front(BlockHeader** list, BlockHeader* block) {
    block->next = *list;
    block->prev = NULL;

    if (*list) {
        (*list)->prev = block;
    }

    *list = block;
}

// Создание аллокатора Best Fit
int createBestFitAllocator(BestFitAllocator* alloc, size_t size) {
    if (!alloc || size < MIN_BLOCK_SIZE * 4)
        return 0;

    // Выделяем память с помощью mmap
    void* memory = mmap(NULL, size,
                        PROT_READ | PROT_WRITE,
                        MAP_PRIVATE | MAP_ANONYMOUS,
                        -1, 0);

    if (memory == MAP_FAILED)
        return 0;

    // Инициализируем структуру аллокатора
    alloc->base = memory;
    alloc->total_size = size;

    // Создаем первый свободный блок на всей выделенной памяти
    BlockHeader* first_block = (BlockHeader*)memory;
    first_block->size = size - HEADER_SIZE;
    first_block->is_free = 1;
    first_block->next = NULL;
    first_block->prev = NULL;

    alloc->free_list = first_block;
    alloc->used_list = NULL;

    return 1;
}

// Выделение памяти по алгоритму Best Fit (наиболее подходящий блок)
void* best_fit_alloc(BestFitAllocator* alloc, size_t size) {
    if (!alloc || size == 0)
        return NULL;

    // Выравниваем запрошенный размер и добавляем место под заголовок
```

```
    size_t needed_size = align_size(size) + HEADER_SIZE;

    // Поиск наиболее подходящего блока (с минимальным достаточным размером)
    BlockHeader* best_block = NULL;
    BlockHeader* current = alloc->free_list;
    size_t best_block_size = SIZE_MAX;

    while (current) {
        if (current->is_free && current->size >= needed_size) {
            // Находим блок с наиболее подходящим размером
            if (current->size < best_block_size) {
                best_block = current;
                best_block_size = current->size;
            }
        }
        current = current->next;
    }

    if (!best_block)
        return NULL;

    // Удаляем найденный блок из списка свободных
    remove_block(&alloc->free_list, best_block);

    // Проверяем, можно ли разделить блок на две части
    if (best_block->size >= needed_size + HEADER_SIZE + MIN_BLOCK_SIZE) {
        // Вычисляем размер остатка после выделения запрошенной памяти
        size_t remaining_size = best_block->size - needed_size;

        // Создаем новый свободный блок из остатка
        BlockHeader* new_free_block = (BlockHeader*)((char*)best_block + needed_size);

        new_free_block->size = remaining_size - HEADER_SIZE;
        new_free_block->is_free = 1;
        new_free_block->next = NULL;
        new_free_block->prev = NULL;

        // Вставляем новый свободный блок в начало списка свободных
        insert_front(&alloc->free_list, new_free_block);

        // Обновляем размер выделенного блока
        best_block->size = needed_size - HEADER_SIZE;
    }

    // Помечаем блок как занятый
    best_block->is_free = 0;

    // Добавляем блок в список занятых
    insert_front(&alloc->used_list, best_block);

    // Возвращаем указатель на данные (пропускаем заголовок)
    return (char*)best_block + HEADER_SIZE;
}
```

```
// Объединение соседних свободных блоков (оптимизированная версия)
static void coalesce_free_blocks(BestFitAllocator* alloc) {
    BlockHeader* current = alloc->free_list;

    while (current && current->next) {
        // Проверяем, являются ли блоки физически соседними в памяти
        if ((char*)current + HEADER_SIZE + current->size == (char*)current->next) {
            // Объединяем текущий блок со следующим
            current->size += HEADER_SIZE + current->next->size;

            // Удаляем следующий блок из списка
            current->next = current->next->next;
            if (current->next)
                current->next->prev = current;

            // Не переходим к следующему блоку, так как текущий блок увеличился
            // и может быть объединен с новым соседом
        } else {
            current = current->next;
        }
    }
}

// Освобождение памяти (оптимизированная версия)
void best_fit_free(BestFitAllocator* alloc, void* ptr) {
    if (!alloc || !ptr)
        return;

    // Получаем указатель на заголовок блока
    BlockHeader* block = (BlockHeader*)((char*)ptr - HEADER_SIZE);

    // Проверяем, не освобожден ли блок уже
    if (block->is_free)
        return;

    // Удаляем блок из списка занятых
    remove_block(&alloc->used_list, block);

    // Помечаем блок как свободный
    block->is_free = 1;

    // Добавляем блок в начало списка свободных
    insert_front(&alloc->free_list, block);

    // Периодически выполняем объединение (не каждый раз!)
    static int free_count = 0;
    if (++free_count % 1000 == 0) {
        coalesce_free_blocks(alloc);
    }
}

// Получение количества свободной памяти
size_t best_fit_free_memory(BestFitAllocator* alloc) {
    if (!alloc)
```

```
        return 0;

    size_t free_memory = 0;
    BlockHeader* current = alloc->free_list;

    // Суммируем размеры всех свободных блоков
    while (current) {
        free_memory += current->size + HEADER_SIZE;
        current = current->next;
    }

    return free_memory;
}

// Уничтожение аллокатора и освобождение памяти
void destroyBestFitAllocator(BestFitAllocator* alloc) {
    if (!alloc)
        return;

    // Освобождаем выделенную память
    munmap(alloc->base, alloc->total_size);

    // Обнуляем указатели
    alloc->base = NULL;
    alloc->total_size = 0;
    alloc->free_list = NULL;
    alloc->used_list = NULL;
}
```

mkc.c

```
#include <stddef.h>
#include <stdint.h>
#include <string.h>
#include <sys/mman.h>

#define PAGE_SIZE 4096U                 // Размер страницы памяти (4KB)
#define MKC_PAGE_FREE  0xFFFFu          // Индикатор свободной страницы
#define MKC_PAGE_LARGE 0xFFFEu          // Индикатор страницы с большим блоком

// Классы размеров для алгоритма McKusick-Karels (степени двойки)
static const size_t mkc_classes[] = {
    16, 32, 64, 128, 256, 512, 1024, 2048
};
#define MKC_NUM_CLASSES (sizeof(mkc_classes)/sizeof(mkc_classes[0]))

// Структура страницы в алгоритме McKusick-Karels
typedef struct MKC_Page {
    uint16_t size_class_index;   // Индекс класса размера или специальный флаг
    uint16_t free_count;         // Количество свободных слотов в странице
    struct MKC_Page* next;       // Указатель на следующую страницу в списке
    uint32_t bitmap[8];          // Битовая карта свободных слотов (256 бит)
} MKC_Page;
```

```c
// Структура аллокатора McKusick-Karels
typedef struct {
    size_t total_size;           // Общий размер выделенной памяти
    void* base_data;             // Указатель на начало области данных
    size_t pages_count;          // Количество страниц в области данных

    MKC_Page* free_pages;        // Список полностью свободных страниц
    MKC_Page* class_pages[MKC_NUM_CLASSES];  // Списки страниц по классам
    MKC_Page* large_pages;       // Список страниц с большими блоками
} MKCAllocator;

// Получение указателя на страницу по её индексу
static inline MKC_Page* page_at(MKCAllocator* alloc, size_t index) {
    return (MKC_Page*)((char*)alloc->base_data + index * PAGE_SIZE);
}

// Очистка битовой карты (все биты устанавливаются в 0)
static inline void bitmap_clear(uint32_t* bitmap) {
    memset(bitmap, 0, sizeof(uint32_t) * 8);
}

// Поиск первого свободного бита в битовой карте
static inline int bitmap_find_free(uint32_t* bitmap, int limit) {
    for (int i = 0; i < limit; i++) {
        // Проверяем, свободен ли i-й бит (0 - свободен, 1 - занят)
        if (!(bitmap[i >> 5] & (1u << (i & 31)))) {
            return i;
        }
    }
    return -1;
}

// Вычисление максимального количества слотов в странице для заданного класса
static inline int max_slots_in_class(int class_index) {
    // Вычисляем сколько слотов помещается в странице (минус заголовок)
    int slots = (PAGE_SIZE - sizeof(MKC_Page)) / mkc_classes[class_index];
    // Ограничиваем 256 слотами для использования 256-битной битовой карты
    return slots > 256 ? 256 : slots;
}

// Определение индекса класса для запрошенного размера
static inline int find_class_index(size_t size) {
    for (int i = 0; i < MKC_NUM_CLASSES; i++) {
        if (size <= mkc_classes[i])
            return i;
    }
    return -1;  // Размер больше максимального класса
}

// Извлечение свободной страницы из списка свободных
static MKC_Page* take_free_page(MKCAllocator* alloc) {
    MKC_Page* page = alloc->free_pages;
    if (!page) return NULL;
```

```
    alloc->free_pages = page->next;
    page->next = NULL;
    bitmap_clear(page->bitmap);
    return page;
}

// Возвращение страницы в список свободных
static void return_free_page(MKCAllocator* alloc, MKC_Page* page) {
    page->size_class_index = MKC_PAGE_FREE;
    page->free_count = 0;
    bitmap_clear(page->bitmap);
    page->next = alloc->free_pages;
    alloc->free_pages = page;
}

// Создание аллокатора McKusick-Karels
MKCAllocator* createMKCAllocator(size_t size) {
    // Минимальный размер - 2 страницы (одна для структуры, одна для данных)
    if (size < PAGE_SIZE * 2)
        return NULL;

    // Выделяем память с помощью mmap
    void* memory = mmap(NULL, size, PROT_READ | PROT_WRITE,
                        MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

    if (memory == MAP_FAILED)
        return NULL;

    // Используем первую страницу для структуры аллокатора
    MKCAllocator* alloc = (MKCAllocator*)memory;
    memset(alloc, 0, sizeof(MKCAllocator));

    alloc->total_size = size;
    alloc->base_data = (char*)memory + PAGE_SIZE;  // Данные начинаются со второй
страницы
    alloc->pages_count = (size - PAGE_SIZE) / PAGE_SIZE;

    // Инициализируем все страницы как свободные
    for (size_t i = 0; i < alloc->pages_count; i++) {
        MKC_Page* page = page_at(alloc, i);
        page->size_class_index = MKC_PAGE_FREE;
        bitmap_clear(page->bitmap);
        page->next = alloc->free_pages;
        alloc->free_pages = page;
    }

    return alloc;
}

// Уничтожение аллокатора McKusick-Karels
void destroyMKCAllocator(MKCAllocator* alloc) {
    if (!alloc) return;
    munmap(alloc, alloc->total_size);
}
```

```
// Выделение памяти с помощью алгоритма McKusick-Karels
void* mkc_alloc(MKCAllocator* alloc, size_t size) {
    if (!alloc || size == 0)
        return NULL;

    // Определяем класс для запрошенного размера
    int class_idx = find_class_index(size);

    // Если размер попадает в один из классов (мелкий блок)
    if (class_idx >= 0) {
        // Ищем страницу с данным классом, в которой есть свободные слоты
        MKC_Page* page = alloc->class_pages[class_idx];
        while (page && page->free_count == 0)
            page = page->next;

        // Если не нашли подходящей страницы, берем новую из свободных
        if (!page) {
            page = take_free_page(alloc);
            if (!page) return NULL;

            // Инициализируем страницу для данного класса
            page->size_class_index = class_idx;
            page->free_count = max_slots_in_class(class_idx);
            page->next = alloc->class_pages[class_idx];
            alloc->class_pages[class_idx] = page;
        }

        // Находим свободный слот в битовой карте
        int slot = bitmap_find_free(page->bitmap, max_slots_in_class(class_idx));
        if (slot < 0) return NULL;

        // Помечаем слот как занятый
        page->bitmap[slot >> 5] |= 1u << (slot & 31);
        page->free_count--;

        // Вычисляем адрес выделенного слота
        return (char*)page + sizeof(MKC_Page) + slot * mkc_classes[class_idx];
    }

    // Большой блок (не помещается ни в один класс)
    size_t pages_needed = (size + PAGE_SIZE - 1) / PAGE_SIZE;
    size_t consecutive_free = 0;
    size_t start_index = 0;
    int found = 0;

    // Ищем последовательность свободных страниц
    for (size_t i = 0; i < alloc->pages_count; i++) {
        MKC_Page* page = page_at(alloc, i);
        if (page->size_class_index == MKC_PAGE_FREE) {
            if (!consecutive_free) start_index = i;
            if (++consecutive_free == pages_needed) {
                found = 1;
                break;
```

```
            }
        } else {
            consecutive_free = 0;
        }
    }

    if (!found) return NULL;

    // Помечаем найденные страницы как занятые для большого блока
    MKC_Page* first_page = page_at(alloc, start_index);
    first_page->size_class_index = MKC_PAGE_LARGE;
    first_page->free_count = pages_needed;

    for (size_t i = 1; i < pages_needed; i++) {
        page_at(alloc, start_index + i)->size_class_index = MKC_PAGE_LARGE;
    }

    // Удаляем эти страницы из списка свободных
    MKC_Page** prev_ptr = &alloc->free_pages;
    while (*prev_ptr) {
        MKC_Page* current = *prev_ptr;
        size_t idx = ((char*)current - (char*)alloc->base_data) / PAGE_SIZE;

        if (idx >= start_index && idx < start_index + pages_needed) {
            // Пропускаем страницу, так как она теперь используется
            *prev_ptr = current->next;
        } else {
            prev_ptr = &current->next;
        }
    }

    // Добавляем первую страницу блока в список больших блоков
    first_page->next = alloc->large_pages;
    alloc->large_pages = first_page;

    return (char*)first_page + sizeof(MKC_Page);
}

// Освобождение памяти в алгоритме McKusick-Karels
void mkc_free(MKCAllocator* alloc, void* ptr) {
    if (!alloc || !ptr)
        return;

    // Определяем, к какой странице принадлежит указатель
    size_t offset = (char*)ptr - (char*)alloc->base_data;
    size_t page_index = offset / PAGE_SIZE;

    if (page_index >= alloc->pages_count)
        return;

    MKC_Page* page = page_at(alloc, page_index);

    // Обработка большого блока
    if (page->size_class_index == MKC_PAGE_LARGE) {
```

```
        size_t pages_count = page->free_count;

        // Освобождаем все страницы большого блока
        for (size_t i = 0; i < pages_count; i++) {
            return_free_page(alloc, page_at(alloc, page_index + i));
        }

        // Удаляем страницу из списка больших блоков
        MKC_Page** prev_ptr = &alloc->large_pages;
        while (*prev_ptr) {
            if (*prev_ptr == page) {
                *prev_ptr = page->next;
                break;
            }
            prev_ptr = &(*prev_ptr)->next;
        }
        return;
    }

    // Обработка мелкого блока
    int class_idx = page->size_class_index;
    size_t class_size = mkc_classes[class_idx];

    // Вычисляем номер слота в странице
    size_t slot = ((char*)ptr - ((char*)page + sizeof(MKC_Page))) / class_size;

    // Проверяем, что слот действительно был занят
    if (!(page->bitmap[slot >> 5] & (1u << (slot & 31))))
        return;

    // Освобождаем слот (устанавливаем бит в 0)
    page->bitmap[slot >> 5] &= ~(1u << (slot & 31));
    page->free_count++;

    // Если страница полностью освободилась, возвращаем её в пул свободных
    if (page->free_count == max_slots_in_class(class_idx)) {
        // Удаляем страницу из списка её класса
        MKC_Page** prev_ptr = &alloc->class_pages[class_idx];
        while (*prev_ptr) {
            if (*prev_ptr == page) {
                *prev_ptr = page->next;
                break;
            }
            prev_ptr = &(*prev_ptr)->next;
        }

        // Возвращаем страницу в пул свободных
        return_free_page(alloc, page);
    }
}

// Получение количества свободной памяти
size_t mkc_free_memory(MKCAllocator* alloc) {
    if (!alloc) return 0;
```

```
    size_t free_memory = 0;

    // Считаем полностью свободные страницы
    for (MKC_Page* page = alloc->free_pages; page; page = page->next)
        free_memory += PAGE_SIZE;

    // Считаем свободные слоты в занятых страницах
    for (size_t i = 0; i < MKC_NUM_CLASSES; i++) {
        MKC_Page* page = alloc->class_pages[i];
        while (page) {
            free_memory += page->free_count * mkc_classes[i];
            page = page->next;
        }
    }

    return free_memory;
}
```

test_allocators.c

```
#define _POSIX_C_SOURCE 200112L
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Подключаем реализации аллокаторов
#include "best_fit.c"
#include "mkc.c"

#define MEMORY_SIZE (4 * 1024 * 1024)   // 4 МБ памяти
#define NUM_OPERATIONS 100000           // Количество операций для теста
#define MAX_BLOCK_SIZE 128              // Максимальный размер блока

// Функция для точного измерения времени
double get_current_time() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC_RAW, &ts);
    return ts.tv_sec + ts.tv_nsec * 1e-9;
}

// Тестирование аллокатора Best Fit
void test_best_fit() {
    printf("=== Тестирование Best Fit аллокатора ===\n");

    BestFitAllocator alloc;
    if (!createBestFitAllocator(&alloc, MEMORY_SIZE)) {
        printf("Ошибка инициализации Best Fit аллокатора\n");
        return;
    }

    // Массив для хранения указателей на выделенные блоки
    void** pointers = malloc(NUM_OPERATIONS * sizeof(void*));
```

```
    // Тест скорости выделения памяти
    double start_time = get_current_time();
    for (int i = 0; i < NUM_OPERATIONS; i++) {
        size_t block_size = rand() % MAX_BLOCK_SIZE + 1;
        pointers[i] = best_fit_alloc(&alloc, block_size);
    }
    double alloc_time = get_current_time() - start_time;
    printf("Выделение %d блоков: %.6f секунд\n", NUM_OPERATIONS, alloc_time);

    // Тест скорости освобождения памяти
    start_time = get_current_time();
    for (int i = 0; i < NUM_OPERATIONS; i++) {
        best_fit_free(&alloc, pointers[i]);
    }
    double free_time = get_current_time() - start_time;
    printf("Освобождение %d блоков: %.6f секунд\n", NUM_OPERATIONS, free_time);

    // Освобождаем аллокатор и создаём заново для теста использования памяти
    destroyBestFitAllocator(&alloc);
    if (!createBestFitAllocator(&alloc, MEMORY_SIZE)) {
        printf("Ошибка переинициализации Best Fit аллокатора\n");
        free(pointers);
        return;
    }

    // Тест эффективности использования памяти
    size_t total_requested = 0;
    for (int i = 0; i < NUM_OPERATIONS; i++) {
        size_t block_size = rand() % MAX_BLOCK_SIZE + 1;
        pointers[i] = best_fit_alloc(&alloc, block_size);
        if (pointers[i]) {
            total_requested += block_size;
        }
    }

    // Вычисление фактора использования памяти
    size_t free_memory = best_fit_free_memory(&alloc);
    double used_memory = (double)(MEMORY_SIZE - free_memory);
    double utilization = used_memory > 0 ?
        (double)total_requested / used_memory * 100.0 : 0.0;

    printf("Фактор использования памяти: %.2f%%\n\n", utilization);

    // Освобождение всех блоков
    for (int i = 0; i < NUM_OPERATIONS; i++) {
        best_fit_free(&alloc, pointers[i]);
    }

    destroyBestFitAllocator(&alloc);
    free(pointers);
}

// Тестирование аллокатора McKusick-Karels
```

```c
void test_mkc() {
    printf("=== Тестирование McKusick-Karels аллокатора ===\n");

    MKCAllocator* alloc = createMKCAllocator(MEMORY_SIZE);
    if (!alloc) {
        printf("Ошибка инициализации MKC аллокатора\n");
        return;
    }

    void** pointers = malloc(NUM_OPERATIONS * sizeof(void*));

    // Тест скорости выделения памяти
    double start_time = get_current_time();
    for (int i = 0; i < NUM_OPERATIONS; i++) {
        size_t block_size = rand() % MAX_BLOCK_SIZE + 1;
        pointers[i] = mkc_alloc(alloc, block_size);
    }
    double alloc_time = get_current_time() - start_time;
    printf("Выделение %d блоков: %.6f секунд\n", NUM_OPERATIONS, alloc_time);

    // Тест скорости освобождения памяти
    start_time = get_current_time();
    for (int i = 0; i < NUM_OPERATIONS; i++) {
        mkc_free(alloc, pointers[i]);
    }
    double free_time = get_current_time() - start_time;
    printf("Освобождение %d блоков: %.6f секунд\n", NUM_OPERATIONS, free_time);

    // Освобождаем аллокатор и создаём заново для теста использования памяти
    destroyMKCAllocator(alloc);
    alloc = createMKCAllocator(MEMORY_SIZE);
    if (!alloc) {
        printf("Ошибка переинициализации MKC аллокатора\n");
        free(pointers);
        return;
    }

    // Тест эффективности использования памяти
    size_t total_requested = 0;
    for (int i = 0; i < NUM_OPERATIONS; i++) {
        size_t block_size = rand() % MAX_BLOCK_SIZE + 1;
        pointers[i] = mkc_alloc(alloc, block_size);
        if (pointers[i]) {
            total_requested += block_size;
        }
    }

    // Вычисление фактора использования памяти
    size_t free_memory = mkc_free_memory(alloc);
    double used_memory = (double)(MEMORY_SIZE - free_memory);
    double utilization = used_memory > 0 ?
        (double)total_requested / used_memory * 100.0 : 0.0;

    printf("Фактор использования памяти: %.2f%%\n\n", utilization);
```

```
    // Освобождение всех блоков
    for (int i = 0; i < NUM_OPERATIONS; i++) {
        mkc_free(alloc, pointers[i]);
    }

    destroyMKCAllocator(alloc);
    free(pointers);
}

int main() {
    srand(1234567);  // Фиксированный seed для воспроизводимости результатов

    printf("Сравнение аллокаторов памяти\n");
    printf("============================\n\n");

    test_best_fit();
    test_mkc();

    return 0;
}
```

Протокол работы программы

```
maxva@LAPTOP-RQH0OLUE:/mnt/d/Programming/OS/КП/src$ ./allocator_test
Сравнение аллокаторов памяти
============================

=== Тестирование Best Fit аллокатора ===
Выделение 100000 блоков: 0.004856 секунд
Освобождение 100000 блоков: 0.003973 секунд
Фактор использования памяти: 48.82%

=== Тестирование McKusick-Karels аллокатора ===
Выделение 100000 блоков: 0.297478 секунд
Освобождение 100000 блоков: 0.002973 секунд
Фактор использования памяти: 72.82%
```

$ strace ./allocator_test

execve("./allocator_test", ["./allocator_test"], 0x7ffce5844e60 /* 38 vars */) = 0

brk(NULL) = 0x5607bdaf8000

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f100f6c5000

access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)

openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

fstat(3, {st_mode=S_IFREG|0644, st_size=37779, ...}) = 0

mmap(NULL, 37779, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f100f6bb000

close(3) = 0

openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220\243\2\0\0\0\0\0"..., 832) = 832

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784

fstat(3, {st_mode=S_IFREG|0755, st_size=2125328, ...}) = 0

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784

mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f100f4a9000

mmap(0x7f100f4d1000, 1605632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f100f4d1000

mmap(0x7f100f659000, 323584, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) = 0x7f100f659000

mmap(0x7f100f6a8000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1fe000) = 0x7f100f6a8000

mmap(0x7f100f6ae000, 52624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f100f6ae000

close(3) = 0

mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f100f4a6000

arch_prctl(ARCH_SET_FS, 0x7f100f4a6740) = 0

set_tid_address(0x7f100f4a6a10) = 28280

set_robust_list(0x7f100f4a6a20, 24) = 0

rseq(0x7f100f4a7060, 0x20, 0, 0x53053053) = 0

mprotect(0x7f100f6a8000, 16384, PROT_READ) = 0

mprotect(0x5607b2699000, 4096, PROT_READ) = 0

mprotect(0x7f100f6fd000, 8192, PROT_READ) = 0

prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0

munmap(0x7f100f6bb000, 37779) = 0

fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x5), ...}) = 0

getrandom("\x63\xad\x57\x9f\x46\x79\x45\x63", 8, GRND_NONBLOCK) = 8

brk(NULL) = 0x5607bdaf8000

brk(0x5607bdb19000) = 0x5607bdb19000

write(1, "\320\241\321\200\320\260\320\262\320\275\320\265\320\275\320\270\320\265 \320\260\320\273\320\273\320\276\320\272\320\260\321"..., 55Сравнение аллокаторов памяти

) = 55

write(1, "============================\n", 29============================

) = 29

write(1, "\n", 1

) = 1

write(1, "=== \320\242\320\265\321\201\321\202\320\270\321\200\320\276\320\262\320\260\320\275 \320\270\320\265 Bes"..., 63=== Тестирование Best Fit аллокатора ===

) = 63

mmap(NULL, 4194304, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f100f0a6000

mmap(NULL, 802816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f100efe2000

write(1, "\320\222\321\213\320\264\320\265\320\273\320\265\320\275\320\270\320\265 100000 \320\261\320\273\320\276"..., 62Выделение 100000 блоков: 0.003010 секунд

) = 62

write(1, "\320\236\321\201\320\262\320\276\320\261\320\276\320\266\320\264\320\265\320\275\320\270\320\265 100000 "..., 68Освобождение 100000 блоков: 0.003394 секунд

) = 68

munmap(0x7f100f0a6000, 4194304) = 0

mmap(NULL, 4194304, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f100f0a6000

write(1, "\320\244\320\260\320\272\321\202\320\276\321\200\320\270\321\201\320\277\320\276\320\273\321\214\320\267\320\276\320\262\320"..., 62Фактор использования памяти: 48.82%

) = 62

munmap(0x7f100f0a6000, 4194304) = 0

munmap(0x7f100efe2000, 802816) = 0

write(1, "=== \320\242\320\265\321\201\321\202\320\270\321\200\320\276\320\262\320\260\320\275\320\270\320\265 McK"..., 70=== Тестирование McKusick-Karels аллокатора ===

) = 70

mmap(NULL, 4194304, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f100f0a6000

brk(0x5607bdbdc000) = 0x5607bdbdc000

write(1, "\320\222\321\213\320\264\320\265\320\273\320\265\320\275\320\270\320\265 100000 \320\261\320\273\320\276"..., 62Выделение 100000 блоков: 0.244932 секунд

) = 62

write(1, "\320\236\321\201\320\262\320\276\320\261\320\276\320\266\320\264\320\265\320\275\320\270\320\265 100000 "..., 68Освобождение 100000 блоков: 0.004772 секунд

) = 68

munmap(0x7f100f0a6000, 4194304) = 0

mmap(NULL, 4194304, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f100f0a6000

write(1, "\320\244\320\260\320\272\321\202\320\276\321\200\320\270\321\201\320\277\320\276\320\273\321\214\320\267\320\276\320\262\320"..., 62Фактор использования памяти: 72.82%

) = 62

munmap(0x7f100f0a6000, 4194304) = 0

```
exit_group(0)                   = ?
+++ exited with 0 +++
```

Вывод

В ходе выполнения курсового проекта были реализованы и протестированы два алгоритма аллокации памяти: Best Fit (наиболее подходящее) и McKusick-Karels. Оба аллокатора предоставляют стандартный интерфейс, аналогичный функциям malloc и free, что делает их удобными для использования в реальных приложениях.

По результатам тестирования можно констатировать, что алгоритмы демонстрируют принципиально разные характеристики, отражающие их внутреннюю логику работы. Best Fit показал выдающуюся скорость операций с памятью: выделение 100000 блоков заняло всего 0.004 секунды, а освобождение — 0.003 секунды. Эта высокая производительность объясняется относительной простотой алгоритма, который использует линейный поиск по списку свободных блоков и минимальные накладные расходы при управлении памятью.

В то же время McKusick-Karels алгоритм оказался значительно медленнее при выделении памяти — 0.316 секунд для того же количества операций, что в 77 раз медленнее Best Fit. Однако при освобождении памяти разница менее заметна: 0.005 секунд против 0.003 у Best Fit. Такое различие в производительности объясняется сложностью внутренней организации McKusick-Karels, который требует определения класса размера, поиска подходящей страницы, работы с битовыми картами и других вспомогательных операций.

Что касается эффективности использования памяти, здесь ситуация обратная. McKusick-Karels продемонстрировал фактор использования 72.82%, в то время как Best Fit — лишь 48.82%. Это существенное различие в 1.5 раза объясняется разными подходами к управлению памятью. Best Fit страдает от внешней фрагментации, когда в памяти накапливаются мелкие свободные блоки, непригодные для больших запросов. McKusick-Karels же, благодаря использованию классов фиксированных размеров и страничной организации, минимизирует фрагментацию и более эффективно использует доступное пространство.

С точки зрения простоты реализации Best Fit явно выигрывает. Его алгоритм основан на классической структуре двусвязного списка с понятной логикой поиска и слияния блоков. McKusick-Karels значительно сложнее, так как требует реализации системы классов размеров, управления битовыми картами, разделения на страницы и обработки больших блоков. Однако эта сложность окупается более предсказуемым поведением и лучшей устойчивостью к фрагментации при длительной работе.

Таким образом, выбор между этими алгоритмами представляет собой классический компромисс в системном программировании. Best Fit предпочтителен в ситуациях, где критична скорость работы с памятью и допустима некоторая неэффективность её использования. McKusick-Karels лучше подходит для систем с

ограниченными ресурсами памяти, где важна максимальная эффективность её использования, даже ценой некоторого снижения производительности.

Оба алгоритма имеют свои области применения: Best Fit может быть оптимальным выбором для высокопроизводительных приложений с интенсивным выделением памяти, в то время как McKusick-Karels лучше подходит для встраиваемых систем и сред с ограниченными ресурсами. В современных операционных системах часто используются гибридные подходы, сочетающие преимущества разных алгоритмов для различных сценариев работы с памятью.